Hybrid Scheduling for Efficient Ray Tracing of Complex Images

Erik Reinhard

Delft University of Technology

Delft, Netherlands

Frederik W. Jansen

Delft University of Technology

Delft, Netherlands

Abstract

Ray tracing is a powerful technique to generate realistic images of 3D scenes. A drawback is its high demand for processing power. Multiprocessing is one way to meet this demand. However, when the models are very large, special attention must be paid to the way the algorithm is parallelised. Combining demand driven and data parallel techniques provides good opportunities to arrive at an efficient scalable algorithm. Which tasks to process demand driven and which data driven, is decided by the data intensity of the task and the amount of data locality (coherence) that will be present in the task. Rays with the same origin and similar directions, such as primary rays and light rays, exhibit much coherence. These rays are therefore traced in demand driven fashion, a bundle at a time. Non-coherent rays are traced data parallel. By combining demand driven and data driven tasks, a good load balance may be achieved, while at the same time spreading the communication evenly across the network. This leads to a scalable and efficient parallel implementation of the ray tracing algorithm.

1   Introduction

From many fields in science and industry, there is an increasing demand for realistic rendering. Architects for example need to have a clear idea of how their designs are going to look in reality. In theatres the lighting aspects of the interior are important too, so these should be modelled as accurately as possible. It is evident that making such designs is an iterative and preferably interactive process. Therefore next to realism, short rendering times are called for in these applications. Another characteristic of such applications is that the models to be rendered are typically very large.

It is very difficult to develop algorithms which meet all these demands. Z-buffering algorithms are known for their speed (they are even suitable for animation purposes), but lack sufficient realism for architectural imagery. On the other hand, those algorithms that do offer an acceptable level of realism, such as ray tracing (see figure 1) and ray tracing based radiosity, are computationally too expensive. Past attempts to overcome these problems generally follow two different approaches. One is to increase the quality of z-buffer algorithms, while retaining their speed. The other method is to speed up radiosity and ray tracing algorithms by using parallel or distributed systems. By having multiple processors in either distributed or shared memory architectures to compute parts of the same problem, considerable speed-ups are to be expected.

The most common way to parallelise ray tracing is the demand driven approach where each processor is assigned a part of the image [1] [2] [3]. Alternatively, coher-
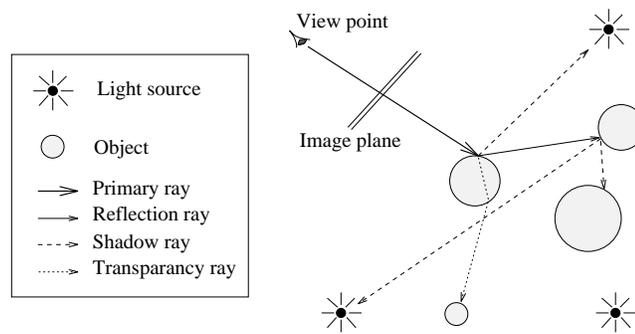
Figure 1: Ray tracing consists of shooting primary rays from the eye point through the screen and tracing reflected and refracted rays recursively if they intersect with objects. At each intersection rays are also shot towards light sources for shadow testing.

ent subtasks may be assigned to processors with a low load. This has the advantage of spreading the load evenly over the processors [4] [5], but it requires either the data to be duplicated with each processor, thereby limiting the model size, or an efficient caching mechanism to be implemented. Caching may reduce the amount of communication, but its efficiency is highly dependent on the amount of coherence between subsequent data requests.

Alternatively, scheduling could be performed by distributing the data over the processors according to a spatial subdivision consisting of voxels. Rays are then traced through a voxel and when a ray enters the next voxel, it is transferred as a task to the processor holding that voxel's objects [7] [8] [9]. This is called the data parallel or data driven approach and it is the basis of our parallel implementation. The advantage of such an approach is that there is virtually no restriction on the size of the model to be rendered. However, there are also some rather severe disadvantages, which include a load balancing problem and a large overhead on communication when there is a large number of processors.

The problems with either pure demand driven or data driven implementations may be overcome by combining the two, yielding a hybrid algorithm [10] [6]. Scherson and Caspary [10] propose to take the ray traversal task, i.e. the intersection of rays with the spatial subdivision structure, such as an octree, to be the demand driven component. As an octree does not occupy much space, it may be replicated with each processor. Provided the load on a processor is sufficiently low, a processor can then perform demand driven ray traversal tasks, in addition to ray tracing through its own voxel in a data driven manner. The demand driven task then compensates for the load balancing problem induced by the data parallel component, while at the same time the amount of data communication is kept low.

Computationally intense tasks that require little data are thus preferably handled in a demand driven manner, while data intensive tasks are better suited to the data parallel approach. A problem with the hybrid algorithm by Scherson and Caspary [10] is that the demand driven component and the data driven component are not well matched, i.e. the ray traversal tasks are computationally not expensive enough to opti-

mally compensate for the load balancing problem of the data parallel part. Therefore, our data parallel algorithm (presented in section 2 and 3) will be modified to incorporate two extra demand driven components differently from Scherson and Caspary [10].

A key notion for our implementation is coherence, which means that rays that have the same origin and almost the same direction, are likely to hit the same objects. Both primary rays and shadow rays directed to area light sources exhibit this coherence. To benefit from coherence, primary rays can be traced in bundles, which are called pyramids in this paper. A pyramid of primary rays has the viewpoint as top of the pyramid and its cross section will be square. First the pyramids are intersected with a spatial subdivision structure, which yields a list of cells containing objects that possibly intersect with the pyramids. Then the individual rays making up the pyramid are intersected with the objects in the cells. By localising the data necessary, these tasks can very well be executed in demand driven mode.

Shadow tracing for area light sources provides another opportunity to exploit coherence, as per intersection, a bundle of rays would be sent towards them. These rays originate from the intersection point and all travel towards the area light. The coherence between these rays is considerable, and they can also be traced in a pyramid, much in the same way primary rays are handled. Further, instead of communicating the shadow ray tasks to neighbouring voxels, shadow tracing can also be performed as a demand driven task at the local processor that found the intersection. This may necessitate fetching object data from other processes, so that caching becomes a relevant technique. However, because the same objects may prove to block a light sources for many possible intersections within a voxel, caching is expected to be highly efficient.

Adding demand driven tasks (processing of primary rays) to the basic data driven algorithm, will improve the load balance and increase the scalability of the architecture.

In this paper, first a description of the data parallel component of our parallel implementation is given in section 2. Next, the demand driven component and the handling of light rays are introduced in section 3. The complete algorithm is described in terms of processes in these sections. A suitable mapping to a processor topology is described in section 4. Finally, conclusions are drawn in section 5.

2   Data parallel ray tracing

To derive a data parallel ray tracer, the algorithm itself must be split up into a number of processes and the object data must be distributed over these processes. Both issues are addressed in this section, beginning with the data distribution.

Objects in a model generally exhibit coherence, which means that objects consist of separate connected pieces bounded in space, and distinct objects are disjoint in this space [4]. Other forms of coherence, such as coherence between rays, where rays with similar origins and directions are likely to intersect the same objects, are derived from this definition. To exploit object coherence in a data parallel algorithm, it is best to store objects that are close together, with the same processor. Such a distribution may
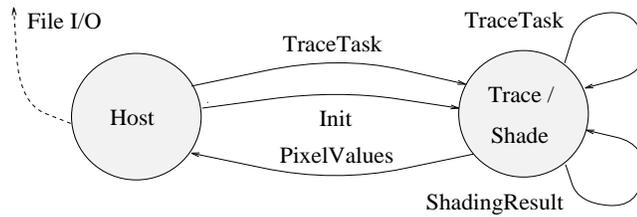
Figure 2: Host and slave processes.

be achieved by splitting the object space into (equal sized) voxels and assigning each voxel with its objects to a process.

Ray tracing is now performed in a master-slave setup. A host process initialises a number of slave processes, and then determines which slave gets which data. After initialisation, for each primary ray the host determines which voxel it originates in and sends this ray as a task to the associated trace process. When all primary rays are dispatched, the host will wait for incoming pixel values, which it will write to an image file. The host and the trace processes and their incoming and outgoing messages are depicted in figure 2.

Each tracing process has the objects pertaining to its voxel. This process reads a ray task from its buffer and traces the ray. Two situations may occur. First, the ray may leave the voxel without intersecting any object. If this happens, the ray is transferred to a neighbouring voxel. Else, an intersection with some locally stored object is found and secondary rays are spawned. These are subsequently traced until one of these two criteria is met.

Shading is performed by the process that found the intersection. To that extend it performs some extra book keeping by storing intersection information. Then the secondary rays are spawned. Somewhere in the future colour values for these secondary rays will be returned, which are stored until all results for an intersection are completed. Then shading is performed and a colour value is returned, see figure 2.

The advantages of this way of parallel ray tracing are that very large models may be rendered as the object database does not need to be duplicated with each process, but can be divided over the distributed memories instead. Ray tasks will have to be transferred to other processes, but as each voxel borders on at most a few other voxels, communication for ray tasks is only local. This means that data parallel algorithms are scalable without too much loss of efficiency.

Disadvantages are that load imbalances may occur. These may be solved by either static load balancing, which most probably yields a suboptimal load balance, or dynamic load balancing, which may induce too much overhead in the form of data communication.

In order to have more control over the average load of each process, and thereby overcome most of the problems associated with data parallel computing, some demand driven components may be added to this algorithm, which is the topic of the following section.
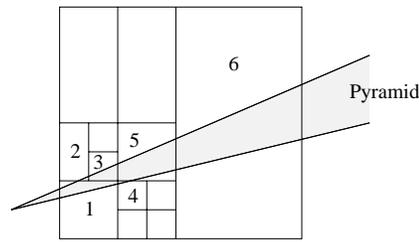
Figure 3: Pyramid traversal generates clip list (cells 1-6).

3   Demand driven pyramid tracing

As tracing rays is by far the most expensive operation in ray tracing, this should be performed as efficiently as possible, i.e. ray coherence should be exploited where present. In ray tracing, different types of rays are generated and each of them has unique properties which call for different handling. We classify rays according to the amount of coherence between them. For example, primary rays all originate from the eye point and travel in similar directions. These rays exhibit much coherence, as they are likely to intersect the same objects. This is also true for shadow rays that are sent towards area light sources.

Restricting ourselves for the moment to primary rays, a common technique to trace a number of rays together, is by replacing them by a generalised ray [12]. Examples are cone tracing [16], beam tracing [17] [18] and pencil tracing [19]). A slightly different method is presented here, consisting of two steps. First, the primary rays are bundled together to form pyramids. These are then intersected with a spatial subdivision. The result of this pyramid traversal is a clip-list, which is an ordered list of cells that intersect (partially) with a pyramid. No object data is necessary to perform this step; only the spatial subdivision structure and the pyramids. Assuming the subdivision structure is a bintree, the process of pyramid traversal is depicted in figure 3.

The second step consists of tracing the individual rays within each pyramid, using the information obtained from the pyramid traversal. Tracing the rays within a pyramid will therefore require a relatively small number of objects, namely the objects that lie within the cells traversed. For this reason, pyramid tracing may be executed in a demand driven manner, as the data communication involved is restricted.

As the pyramids diverge the further they are away from the origin, coherence decreases. There are two other causes for the decrease in coherence. Some of the rays within a pyramid may intersect with an object, leaving gaps between the primary rays that continue. Also, intersections cause secondary rays that may travel in completely different directions. It may therefore be necessary after a certain distance is travelled, to switch from demand driven execution to data driven execution, as described in the preceding section.

Another opportunity to exploit ray coherence, is provided by shadow rays. Whereas in data parallel tracing, the voxels that contain light sources, may become bottlenecks, a demand driven approach may successfully circumvent this problem. Especially area
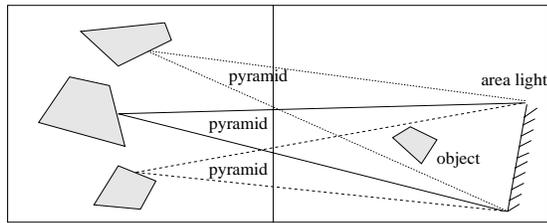
Figure 4: The object in the right voxel is needed for light pyramid tracing by the process managing the left voxel.
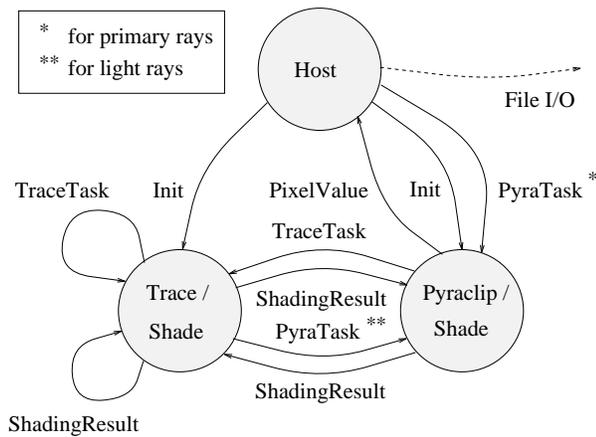

Figure 5: Host, trace and pyraclip tasks and the communication between them.

light sources, which generate a bundle of rays per intersection, are problematic in data parallel tracing. It is therefore advantageous to apply some form of pyramid tracing to these rays as well. To avoid contention at the processes containing light sources, light pyramids may be processed locally by the processes that initiated them [14]. If we choose to trace shadow pyramids with the process that spawned them, it may be necessary to fetch objects from remote processes, as figure 4 illustrates. In this figure, the object in the right voxel, which is in front of the area light source, is needed by all light pyramids depicted.

Each process that may spawn light pyramids should be equipped with a cache, which reduces data communication. The effectiveness of such a cache is estimated to be high, because many pyramids generated from within a voxel, will intersect with the same objects.

In figure 5, the resulting setup of host, data parallel trace and demand driven pyraclip tasks is depicted. Pyraclip tasks generated by the host are for primary rays and pyraclip tasks originating from a trace process are for tracing shadow rays.

4   Architecture and implementation

Up until this point, the algorithm is described in terms of processes, which is architecture independent. If this algorithm is to be implemented, these processes must be mapped onto a certain hardware architecture.

There is a choice between message passing and shared memory architectures. Message passing architectures are naturally scalable, but communication between processes must be explicitly specified by the application. To reduce the amount of communication, caching may be used. The performance of the cache depends strongly on the amount of data locality. Similar arguments apply for advanced shared memory architectures that also use local caching to reduce the amount of communication with main memory [15]. Therefore, our algorithm is suitable for both types of architecture, as it is designed to minimise the amount of data needed per intersection. Our current system is implemented on a message passing architecture, the mesh connected Parsytec GCel.

The algorithm distinguishes a number of different processes. First of all, there is a host process, which initiates pyramid traversal tasks and receives pixel values. Second, there are pyramid tracing processes which operate in demand driven mode. Ray tracing processes for tracing individual secondary rays form a third category. These processes also perform light pyramid tracing, using a cache for objects. Shading is performed by the process that found the intersection.

One possible mapping of processes onto processors is depicted in figure 6. There is one host processor and a cluster of slave processors. Each of these slave processors runs data parallel tracing processes. Some of them, typically the ones with a low load, may also run a pyramid tracing process. Pyramid tracing tasks should preferably be scheduled to processors that are located close to the host processor, in order to minimise long distance communication.

In a data parallel tracing cluster, the optimal number of processors will be bounded by the amount of task communication which increases when new processors are added. If this optimal number of processors is lower than the number of processors available for data parallel tracing, it may be possible to add other identical data parallel clusters. In that case the object database is distributed within a cluster, but duplicated over the clusters. Ray tasks may then be executed in any of the available clusters, without ever having to be transferred between clusters. This scheme of cluster replication works in ray tracing by virtue of the static nature of the object database, but not in radiosity and other algorithms, such as Radiance [20], which update the object database in the course of execution.

For the current implementation we have used an adapted version of Rayshade [21], which is a sequential public domain ray tracer. The language used is C, extended with the PVM parallel library [22]. Although using standard libraries may be less efficient than hard coding for a specific architecture, it has great advantages in terms of portability.

Currently, the demand driven tasks consist of primary rays only. Scheduling is performed statically and there are three scheduling criteria. These are processors with a voxel on the model boundary, processors with a low number of objects and processors
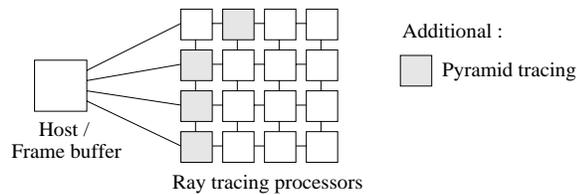
Figure 6: Mapping of processes onto processors

with a low number of light sources.

5 Conclusions

The two basic approaches to parallel rendering, demand driven and data parallel, both have their advantages and shortcomings. Data driven scheduling allows the algorithm to be scaled, but almost invariably leads to load imbalances. Demand driven approaches achieve better load balancing, but may suffer from a communication bottleneck due to insufficient cache performance, which makes them less scalable.

In order to arrive at an efficient scalable algorithm for ray tracing, the algorithm is best split into a demand driven and a data parallel component. Rays that show much coherence are then most efficiently traced by a demand driven algorithm which exploits ray coherence as much as possible. The pyramid tracing algorithm described in this paper does precisely that.

For rays that are less coherent, such as reflection rays and transparency rays, the data parallel approach pays off, as fetching object data, which is associated with demand driven solutions, is too expensive for single rays.

As both message passing and shared memory systems rely on caching mechanisms for efficiency, an application running on such machines should optimise data locality. By exploiting ray and object coherence present in ray tracing, data locality can be preserved. This holds for simple ray tracing applications, but also for more involved ray tracing based radiosity and Monte Carlo algorithms, where for each patch or intersection, a hemisphere is sampled.

Features to be implemented next include the demand driven scheduling of light rays and the implementation of dynamic scheduling for pyramid tasks.

References

[1] D. J. Plunkett, M. J. Bailey, The vectorization of a ray-tracing algorithm for improved execution speed, IEEE Computer Graphics and Applications **5**(8), 52–60, (1985).

[2] F. C. Crow, G. Demos, J. Hardy, J. McLaugglin, K. Sims, 3d image synthesis on the connection machine, *in* Proceedings Parallel Processing for Computer Vision and Display, Leeds, (1988).

[3] T. T. Y. Lin, M. Slater, Stochastic ray tracing using SIMD processor arrays, The Visual Computer **7**(4), 187–199, (1991).

[4] S. A. Green, D. J. Paddon, Exploiting coherence for multiprocessor ray tracing, IEEE Computer Graphics and Applications **9**(6), 12–27, (1989).

[5] L. S. Shen, A Parallel Image Rendering Algorithm and Architecture Based on Ray Tracing and Radiosity Shading, PhD thesis, TUDelft, Delft, (1993).

[6] F. W. Jansen, A. Chalmers, Realism in real time?, *in* 4th EG Workshop on Rendering, pp. 1–20, (1993).

[7] M. A. Z. Dippé, J. Swensen, An adaptive subdivision algorithm and parallel architecture for realistic image synthesis, ACM Computer Graphics **18**(3), 149–158, (1984).

[8] J. G. Cleary, B. M. Wyvill, G. M. Birtwistle, R. Vatti, Multiprocessor ray tracing, Computer Graphics Forum, **5**(4), 3–12, (1986).

[9] H. Kobayashi, S. Nishimura, H. Kubota, T. Nakamura, Y. Shigei, Load balancing strategies for a parallel ray-tracing system based on constant subdivision, The Visual Computer **4**(4), 197–209, (1988).

[10] I. D. Scherson, C. Caspary, A self-balanced parallel ray-tracing algorithm, *in* P. M. Dew, R. A. Earnshaw, T. R. Heywood, eds, Parallel Processing for Computer Vision and Display, Vol. **4**, Addison-Wesley Publishing Company, Wokingham, pp. 188–196, (1988).

[11] T. Whitted, An improved illumination model for shaded display, Communications of the ACM **23**(6), 343–349, (1980).

[12] A. S. Glassner, ed., An Introduction to Ray Tracing, Academic Press, San Diego, (1989).

[13] G. J. Ward, F. M. Rubinstein, R. D. Clear, A ray tracing solution for diffuse interreflection, ACM Computer Graphics **22**(4), 85–92, (1994).

[14] T. Priol, K. Bouatouch, Static Load Balancing for a Parallel Ray Tracing on a MIMD Hypercube, The Visual Computer, **5**(12), 109–119, (1989).

[15] J. S. Singh, A. Gupta, M. Levoy, Parallel Visualization Algorithms: Performance and Architectural Implications, IEEE Computer, **27**(7), 45-55, (1994).

[16] J. Amanatides, Ray tracing with cones, ACM Computer Graphics **18**(3), 129–135, (1984).

[17] P. S. Heckbert, P. Hanrahan, Beam tracing polygonal objects, ACM Computer Graphics **18**(3), 119–127, (1984).

[18] N. Greene, Detecting intersection of a rectengualr solid and a convex polyhedron, *in* P. S. Heckbert, ed., Graphics Gems IV, AP Professional, Cambridge, MA, chapter 1.7, pp. 74–82, (1994).

[19] M. Shinya, T. Takahashi, S. Naito, Principles and applications of pencil tracing, ACM Computer Graphics **21**(4), 45-54 (1987).

[20] G. J. Ward, The radiance lighting simulation and rendering system, ACM Computer Graphics pp. 459–472. SIGGRAPH '94 Proceedings, (1994).

[21] C. E. Kolb, Rayshade User's Guide and Reference Manual. Included in Rayshade distribution, which is available by ftp from princeton.edu:pub/Graphics/rayshade.4.0, (1992).

[22] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, PVM 3 Users Guide and Reference Manual, Oak Ridge National Laboratory, Oak Ridge, Tennessee. Included with the PVM 3 distribution, (1993).